A Study of Techniques for Introducing Concurrency into List Processing Library Routines in Modern Concurrent Functional Programming Languages

Brent Heeringa
Student
Division of Science and Mathematics
Computer Science Discipline
University of Minnesota, Morris
heerinba@cda.mrs.umn.edu

Scott Lewandowski
Professor of Computer Science
Division of Science and Mathematics
Computer Science Discipline
University of Minnesota, Morris
swl@cda.mrs.umn.edu

Abstract

Erlang is a modern functional programming language with additional features that support explicit concurrency through lightweight processes and message passing. Erlang's clean semantics and lack of side effects make it possible to develop high quality software that can easily take advantage of the availability of multiple processors. Like most functional programming languages, Erlang provides a number of library routines (e.g. **`map`**) for processing and manipulating lists. These routines are an excellent target for the introduction of concurrent processing techniques. In this work we explore various methods for introducing concurrency into list processing routines such as **`map`**. We report on the results of our experiments using the Erlang Development Environment, and discuss which approaches to concurrency are most beneficial given the number of processing nodes available and the properties of the computation (e.g. type of function being applied, size of the input list, etc.).

Introduction

Functional languages have long been favored for the ease they provide in processing lists of information. Early functional languages, however, were often criticized as being too inefficient. Advances in implementation techniques (such as graph reduction) and hardware platforms have made it possible in recent years to design and implement efficient functional programming languages. Many of these languages (e.g. Concurrent Clean, Haskell, Erlang) provide all of the features of more traditional functional languages (e.g. Lisp) as well as extensions for parallel and distributed processing and/or object-oriented programming [Wilhelm 1995, Plasmeijer 1997]. As efficiency has improved, more and more people have come to recognize the power inherent in modern functional programming languages. Their clean semantics and lack of side effects make it possible to develop high quality software systems that can easily take advantage of distributed processing resources. Modern functional languages are increasingly finding acceptance as a viable software tool for building solutions to real world problems. Erlang, for example, was designed as a language for programming large industrial telecommunications switching systems. It is also suitable for programming embedded real-time control systems [Armstrong et al. 1996].

A key data type in functional languages is the list. These languages typically provide a number of library routines for building, processing, and manipulating lists. Oddly enough, even in modern functional languages that support concurrency, little progress has been made in including concurrent versions of list processing routines (such as map or sort) into language libraries so that the efficiency benefits of concurrency can be easily taken advantage of in building solutions to problems.

Erlang (from Ericsson Telecommunications Systems) is a modern functional programming language that provides built in language support for concurrency and which includes in its libraries a concurrent version of the map function which applies a specified function to all elements in a list. In the next section we describe the map function and our variations in more detail. We then describe our experiments and discuss our experimental results. Finally, we discuss our plans for future work and enumerate the conclusions we have reached. A description of the Erlang language focusing specifically on the features it provides for supporting concurrent computations as well as specific implementation details of our parallel functions can be found in the appendices.

The Map Function

Most functional programming languages provide the ability to apply a function to every item in a list. This function is typically called **`map`**, and for the purposes of this discussion will be assumed to take two arguments: a function of one argument, and a list. The function is applied in turn to each of the elements of the list and the results of these function applications are collected in a list. Put another way:

$$\texttt{map(f, [}l_1, l_2, \ldots, l_n\texttt{])} \Rightarrow \texttt{[f(}l_1\texttt{), f(}l_2\texttt{), }\ldots\texttt{, f(}l_n\texttt{)]}$$

where **`f`** is a function of one argument, and where the elements of the list are of the appropriate type for **`f`**. For example:

`map(sqr, [1, 3, 5, 7 ]) ⇒ [1, 9, 25, 49]`.

Map can be implemented as a simple linear recursion over the elements of the list **`l`**. In Erlang, we would express this is as follows:

```
map(F, [H|T]) → [apply(F, [H])|map(F, T)];
map(F, []) → [].
```

Erlang uses pattern matching, a powerful tool for binding values to terms of the same shape. Pattern matching allows a function to be expressed as a collection of case definitions. When defining a function, the order of the case definitions is unimportant due to pattern matching. For example, the base case of map (i.e. the declarative containing the empty list as the second argument) may be positioned last because the pattern for the empty list will in fact be matched when appropriate.

In Erlang there are several patterns which are useful when creating functions that process or manipulate lists:

- **`[]`** denotes the empty list.
- **`[H]`** denotes a list containing exactly one term, where **`H`** is bound to the value of that term
- **`[H|T]`** denotes a list containing at least one term, where **`H`** is bound to the value of the first term in the list, and **`T`** is bound to the rest of the list (note that **`T`** will always be a list and that **`T`** may be bound to the empty list).

When used in a context similar to the right hand side of the first case definition above, the Erlang operator **`|`** indicates concatenation (i.e. the left operand to **`|`** is concatenated to, or added to the front of, the list denoted by the right hand operator to **`|`**). Thus **`[1|[9, 25]] ⇒ [1, 9, 25]`**. The apply routine calls the function denoted by its first argument on the elements of the list it is given as its second argument. Thus **`apply(max, [14, 7])`** is equivalent to **`max(14, 7)`**. Note that the length of the list and the arity of the function must be the same.

Tracing through the example above results in the following:

```
   map(sqr, [1, 3, 5, 7])
⇒  [apply(sqr, [1])|map(sqr, [3, 5, 7])]
⇒  [apply(sqr, [1])|[apply(sqr, [3])|map(sqr, [5, 7])]]
⇒  [apply(sqr, [1])|[apply(sqr, [3])|[apply(sqr, [5]|map(sqr, [7])]]]
⇒  [apply(sqr, [1])|[apply(sqr, [3])|[apply(sqr, [5]|
   [apply(sqr, [7])|map(sqr, [])]]]]
⇒  [apply(sqr, [1])|[apply(sqr, [3])|[apply(sqr, [5]|
   [apply(sqr, [7])| [ ] ]]]]
⇒  [sqr(1)|[sqr(3)|[sqr(5)|[sqr(7)|[]]]]]
⇒  [1|[9|[25|[49|[]]]]]
⇒  [1, 9, 25, 49]
```

Since modern functional languages are (typically) side effect free, each application of the function **`f`** to an element of **`l`** is independent of all other applications of **`f`**. Therefore, as long as the results are assembled appropriately, the order of application is irrelevant. This makes **`map`** a prime target for the introduction of concurrency.

The Erlang library provides a straightforward concurrent implementation of the map function, called **`pmap`** (i.e. parallel map), where each application of **`f`** to an element of **`l`** is computed within its own process (i.e. a separate, self-contained unit of computation). With a sufficiently computationally intensive function and enough processing nodes this approach yields significant performance gains over the sequential implementation. There is however significant overhead associated with this approach – specifically the costs of spawning a potentially large number of processes and of sending a potentially large number of messages over the communications network.

In our experiments we investigate the costs and benefits associated with concurrent implementations of the map function. We examine the performance of the **`pmap`** routine provided in the Erlang library as well as that of two implementations of our own which we briefly describe below. Both of our implementations borrow a technique commonly used in implementations of quick and merge sorts. When implementing these sorting algorithms, lists aren't always broken down to single elements due to the overhead associated with recursion. Rather, the lists are broken down to some pre-determined threshold length, at which point a simpler (non-recursive) sorting algorithm (e.g. insertion sort) is used. We observe that in building concurrent implementations of the list function map it is not necessary to spawn a new process for each list element. Overhead costs can be significantly reduced by breaking the list down to some pre-determined threshold length, at which point the sequential version of map which is discussed above can be used.

Our first approach, which we'll refer to as **`smap`** (i.e. map over sub-lists), is a straightforward modification of the Erlang library routine **`pmap`**. In this implementation we do not create a new process for each application of **`f`** to an element of **`l`**. Instead, we divide the list into segments of length eight or less and create a process for each segment wherein the sequential version of **`map`** is used to apply **`f`** to each of the eight (or fewer) list elements. Since fewer processes are created using this approach there is less of an overhead penalty compared to the standard Erlang approach.

In this approach, all of the work involved with splitting the list into segments occurs in one process which we'll refer to as the root process. A new child process is then spawned for each list segment of length eight or less. This leads to a process organization similar to that shown in Figure 1. The root process is responsible for reassembling the results from all of the child processes into the final result. For a list of N elements, this approach requires the creation of $\lceil N / 8 \rceil + 1$ processes.

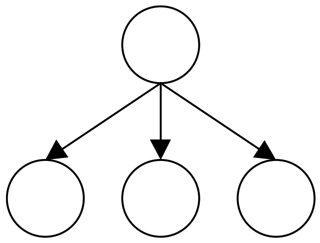

Figure 1: Process organization in **smap**

Our second approach, which we'll refer to as **tmap** (i.e. map using a tree split), also splits the list into segments of length eight or less. Like **smap** it creates a process for each segment wherein the sequential version of **map** is used to apply **f** to each of the eight (or fewer) list elements. The key difference when compared with the previous approach involves how the list is split up and where the work occurs.

In this approach we use a binary splitting technique that leads to a hierarchical process organization similar to that shown in Figure 2. As was the case with **smap**, the work performed by the leaf processes is that of applying the sequential version of **map** to lists of length eight or less. Internal processes (shown in gray) are responsible for splitting their input list in half, for spawning two child processes to operate on each half of the list, and for reassembling the results from those child processes into a result. This result is returned back to the node's parent process. For a list of *N* elements, this approach requires the creation of no more than $2^{\lceil \log_2(\lceil N/8 \rceil) \rceil + 1}$ - 1 processes.

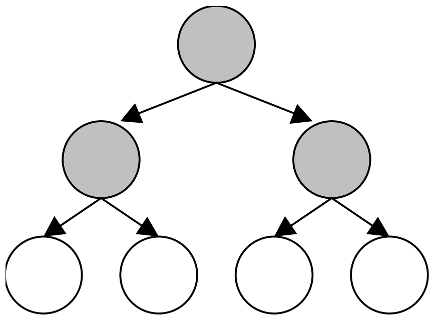

Figure 2: Process organization in tmap

Our Experiments

We implemented both approaches described above as Erlang routines and ran a number of experiments mapping functions of various orders of magnitude over lists of varying sizes. Table 1 summarizes the experiments that we report on in the next section.

	O(1)	O(N)	$O(N^2)$	$O(N^3)$
map	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384
pmap	16, 128, 1024, 16384	16, 128, 1024, 16384	16, 128, 1024, 16384	16, 128, 1024, 16384
smap	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384
tmap	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384, 32768	16, 128, 1024, 16384

Table 1: Experiment Summary

To investigate what effect increasing the number of Erlang nodes has on our distributed routines, experiments were run on configurations of two to five nodes. Each node was started on its own workstation – either on one of two Pentium II 300 MHz systems with 128 Mb of RAM or on one of three Pentium 200 MHz systems with 64 Mb of RAM. For each configuration of nodes, mapping routine, mapped function, and list size, we performed a series of twelve runs. In reporting our figures, we eliminated the fastest and slowest of the twelve runs and averaged the results of the remaining ten.

Note that the upper limit on the size of lists used in our experiments is limited in the case of **`pmap`**. This is due to a limit on the number of processes the free version of the Erlang Development Environment will allow at any given moment.

Our Results

The results of our experiments were, by and large, what we expected, with one small caveat. The graph shown in Figure 3 below is representative of our overall findings. Note that nodes *one* through *three* are the Pentium 200 MHz systems with 64 Mb of RAM.

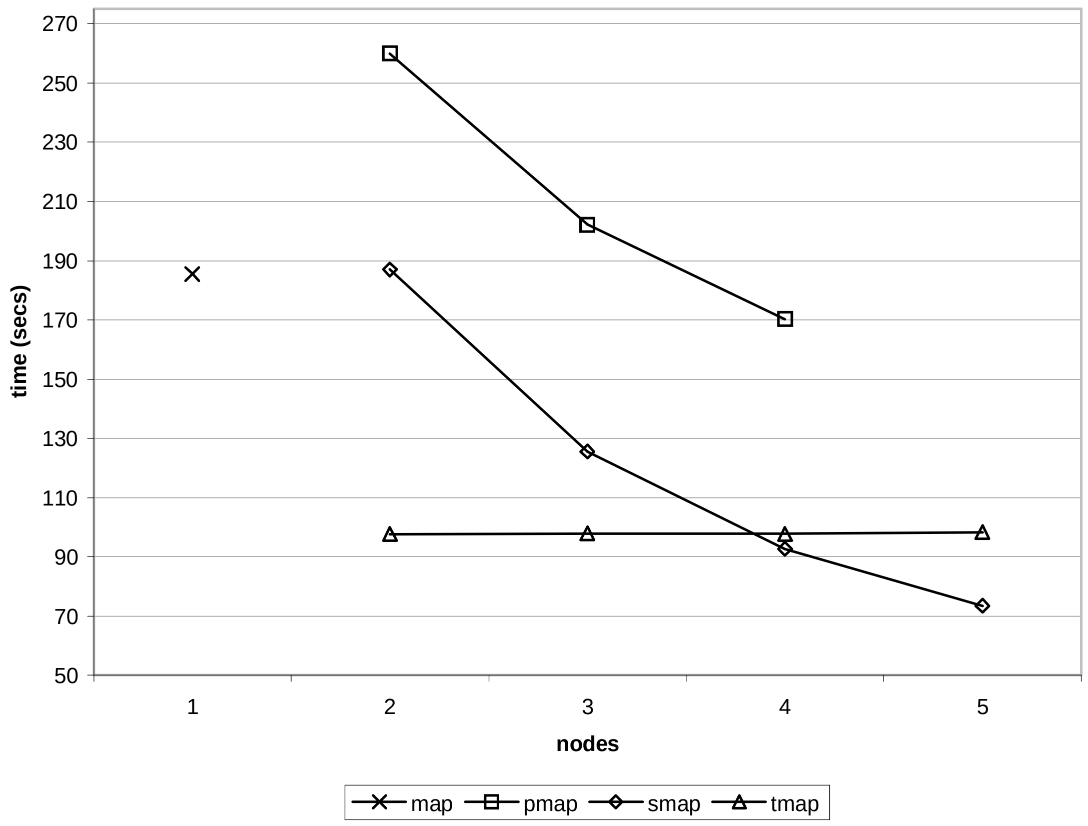

Figure 3: Results of mapping an $O(N^3)$ function onto a 16384 element list

The single cross in Figure 3 indicates the execution time of the sequential version of **`map`**. As discussed above, the performance of the version of **`pmap`** provided in the Erlang library suffers due to the significant overhead involved in spawning 16384 processes. The performance of **`smap`** takes a small initial hit due to overhead when only two nodes are involved, but quickly and dramatically improves as more processing nodes are added. While we are still investigating the reasons for **`tmap`**’s initial but dramatic performance gains, we do have an explanation for its lack of improvement as more processing nodes are added. The problem lies in how the Erlang library routine **`parallel_eval`** (which lies at the heart of all the distributed versions of the map function) assigns processes to nodes. Basically, no matter how many processing nodes are available, **`tmap`** as implemented, will only ever use two of them (see the appendices for more details on the inner workings of our code and the Erlang library routines). The idea that the work is split evenly between two nodes at least partially explains why the performance of tmap in this instance is roughly half that of the sequential version of **`map`**.

Figure 4 shows the results of mapping an O(N^2) function onto a 32768 element list. The performance trends are consistent with those shown in Figure 3, although **`tmap`** displays an unusual improvement in its performance with the addition of a fifth node. This cannot be attributed to a different subset of nodes being used, as the experimental results indicate the same nodes would have been used in either case. Since the time difference is only a matter of seconds, a probable explanation for **`tmap`**'s improved performance, is a decrease in network traffic flow. Note also the graph in Figure 4 does not display the results for **`pmap`** as they are significantly higher than the times shown, ranging from 68 seconds using five nodes up to 76 seconds using two nodes.

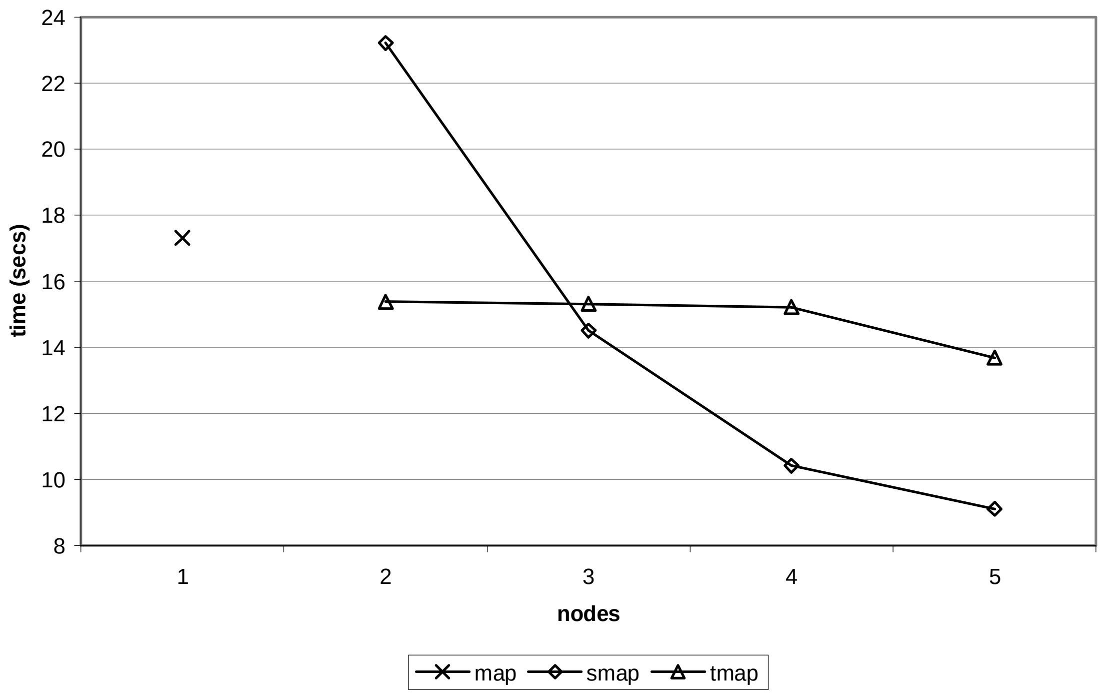

Figure 4: Results of mapping an O(N^2) function onto a 32768 element list

Figure 5 illustrates the consistent behavior of **`smap`** over all of the experiments outlined in the previous section. Notice the recurring pattern in each set of four columns; as more processing nodes are added, **`smap`** almost invariably produces results more quickly. Additional experiments, varying the size of the sublists (i.e. using values other than eight), could help determine an optimal relationship between sublist length and performance.

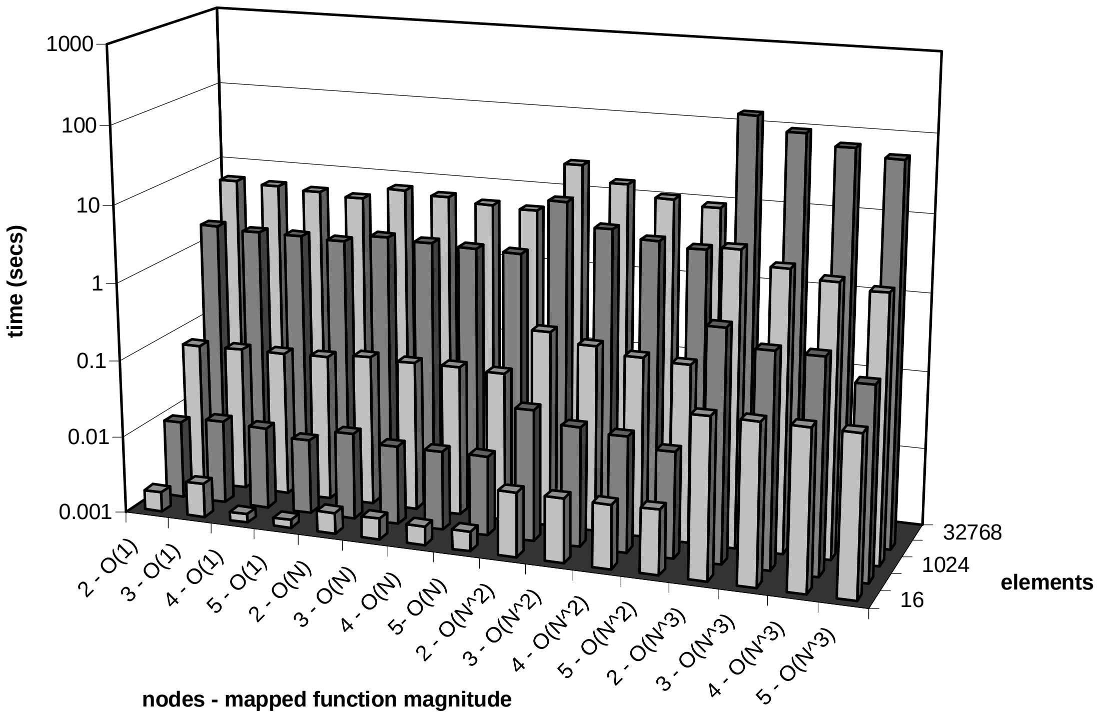

Figure 5: **`smap`** performance summary

Future Work

One unsurprising result of this research found that parallel computations in Erlang, using the generic **`parallel_eval`** function, were computationally bound by the slowest processor. This is due to the fact that processes are distributed evenly in a round robin fashion (i.e. job *j* is assigned to node *n mod j*). Given the heterogeneous nature of computing equipment commonly available even within a single lab, we would ideally like to create concurrent routines where faster processors receive a larger portion of the computation. This would replace the even distribution of work currently supported, and allow for even further performance gains.

Erlang provides some simple mechanisms to help with load distribution including the notion of a node pool that is used in conjunction with a load checking algorithm. The Erlang libraries provide a function called **`statistic_collector`** which performs load checks on nodes in the pool by simply checking the length of the run queue. The run queue is “simply the number of processes which are ready to run” [Armstrong et al. 1996]. Collecting statistics at run time creates an adaptive load distribution which could provide a noticeable speed increase, as well as solve many of the problems associated with the round robin style of process distribution. Notably, the failure of **`tmap`** to spawn processes beyond the scope of two nodes is solved by adaptive load distribution.

Another interesting tool might be the development of a concurrent evaluator, that is, an algorithm that may determine when concurrency brings about benefits, and conversely when evaluating a function locally is more beneficial. Because of the transparent distribution provided by Erlang, an implementation of such an evaluator function may fit nicely into list processing library routines. For example, the algorithm might check number of nodes available, current run-queue statistics on those nodes, and finally the length of arguments given to the possible concurrent routine as input. The evaluator would be folded into possible concurrent function definitions, as opposed to a standalone function, since its role would not be to determine whether the function itself would benefit from concurrency, but rather if the current computing situation would be beneficial. That is, the evaluator would not determine whether function **`foo`** would benefit from concurrency, but rather, if the current environment (i.e., number of nodes, current load, etc. . .) lends itself well to execution of the function in parallel.

Finally, although this research concentrated on parallelizing one common list processing routine–**`map`**, it would be trivial to apply such parallelization across many other library functions. For example, common sorting routines such as mergesort and quicksort could reap the benefits of concurrency quite easily if implemented in a distributed manner. Other functions which perform operations over the elements of a large lists, such as **`all`**–a function which takes a predicate and a list as arguments and determines whether each element in the list satisfies the predicate–may be also be viable candidates [Springer 1989].

Conclusions

The research conducted within the scope of this paper suggests that the addition of concurrent list processing routines in standard modern functional programming libraries would be beneficial. The Erlang shell provides a concurrent-friendly environment where implementation of such parallel routines may take place. Our research demonstrates the ability to easily improve upon current parallel library functions. Our implementation of **`smap`**, for instance, outperformed the simple **`pmap`** in all computational experiments. The research also indicates that although using the round robin method of load distribution produces good results, the incorporation of adaptive load distribution techniques would provide even further performance benefits.

Appendix 0: Details of Distributed Erlang as Related to List Processing

Appendix 0 is provided for the reader interested in details related to distributed Erlang and its role in constructing parallel versions of list processing routines.

Erlang provides transparent distribution; that is, the ability to spawn remote procedure calls without general knowledge of other Erlang nodes. An Erlang node is a self-contained unit; an entity that may run standalone or as part of a network of nodes. Not surprisingly, a single machine may run multiple nodes. Alternatively, an Erlang node may be thought of as an instance of the Erlang shell with a unique name identifier. The shell is basically a command line interpreter. Name identifiers are given by the user upon initiation of the Erlang shell and are of the form 'nodename@domain.name'. Nodes with identical magic cookies (a simple, authority driven, security mechanism where a string of alphanumeric characters, called the cookie, is passed along with every remote procedure call) are allowed communication through TCP/IP. Distribution is thus achieved through lightweight asynchronous message passing between nodes.

"Many operating systems provide complex mechanisms such as remote procedure calls, global name servers, etc. as components of their systems. Erlang, however, provides simpler primitives from which such mechanisms can be constructed" [Armstrong et al. 1996]. Implementations of global name and remote procedure servers are included in current releases of the Erlang Development Environment. In this work, we use the supplied remote procedure server in order to specify which nodes receive what processes.

Included in the Erlang libraries are two important modules – **`parallel_eval`** and **`promise`**. The first function provides an abstraction for parallel evaluation and the second, a means by which distribution and concurrency can occur without idle wait time.

Erlang's **`parallel_eval`** function takes as an argument a list of 3-tuples. Every 3-tuples consists of a module name, function name, and a list of arguments for the given function. The Erlang syntax for this list of tuples is shown below.

[{mod_name, function, [arg$_1$, arg$_2$, arg$_3$, . . ., arg$_n$]} , . . . ,

{mod_name, function, [arg$_1$, arg$_2$, arg$_3$, . . ., arg$_n$]}]

The purpose of the argument list is to provide **parallel_eval** with a pre-determined collection of tasks to be evaluated in parallel. **parallel_eval**, on the most basic level, assigns these collections of sub-tuples to nodes, executing remote procedure calls on those nodes, using the module/function given in each 3-tuple. **parallel_eval** also preserves the order of the collections given as arguments. Purposefully, **parallel_eval** is implemented in such a manner that the programmer decides, through splitting, how modules, functions, and data will be divided into collections.

The definition of **parallel_eval** is as follows:

```
parallel_eval(ArgL) ->
      Nodes = [node() | nodes()],
      Keys = map_nodes(ArgL, Nodes, Nodes),
      lists:map({promise,yield}, [], Keys).

map_nodes([], _, _) ->
      [];
map_nodes(ArgL, [], Orig) ->
      map_nodes(ArgL, Orig, Orig);
map_nodes([{M, F, A}|Tail], [Node|MoreNodes], Orig) ->
      [promise:call(Node, M, F, A) | map_nodes(Tail, MoreNodes, Orig)].
```

One can see from the function definition that **parallel_eval** gathers a list of nodes using the **nodes()**function, appends itself (i.e. through the call to **node()**) onto the list, and consequently generates a list of promises through **map_nodes**, using the argument list and available nodes as input.

map_nodes recurses through the argument list, binding the module, function and argument list, of each sub-tuple to variables **M**, **F**, and **A** respectively. It also binds **Node** to the first term of the node list. It then constructs a list of function calls to **promise:call** with the previous bound variables using a recursive call with arguments **Tail**, **MoreNodes**, and **Orig**. It is important to note that when the node list becomes empty, it is matched with the second case definition of **map_nodes**. This declaration then calls **map_nodes** with the original node list **Orig** that conveniently has been passed along by the function, creating a round robin style of node distribution. Finally, the **yield** function is mapped to the list of promises generated by **map_nodes** (which has been bound to the variable **Keys**).

A promise solves many problems related to synchronous message passing in that it provides a place holder for returned remote procedure call values, successfully eliminating the need to wait for a process to return in order to continue with evaluation of an algorithm. The Erlang promise module is written as follows:

```
-module(promise).

call(Node, Mod, Fun, Args) ->
      spawn(promise, do_call, [self(), Node, Mod, Fun, Args]).

yield(Key) ->
    receive
        {Key, {promise_reply, R}} ->
            R
    end.

 do_call(ReplyTo,N,M,F,A) ->
      R = rpc:call(N,M,F,A),
      ReplyTo ! {self(), {promise_reply, R}}.
```

The promise module while small, provides important support for efficient distributed computations. The **call** function spawns as a local process **do_call**, providing the caller's process identification number (the call to **self()** returns the PID), the name of the node on which to execute, the module and function to execute, and an argument list as arguments. Note that Erlang's spawn function is analogous to UNIX's "&" in that it returns a PID. **do_call** in turn uses the remote procedure server to evaluate function **F** in module **M** over the argument list **A** on node **N** producing result **R**. It subsequently sends a message to itself (using the "!" operator) which includes the result **R**.

The idea of a promise, or simply, a place holder, results from this explicit message passing. The function **yield** will return **R**, the desired result, only when it has been informed by **do_call** that **R** has been evaluated. Thus, **yield** will only finish evaluating when it receives the proper message, in essence blocking advancement of the function till a message is sent to it. One can see from the above definition of **parallel_eval** that wrapping **yield** around a function call to **promise:call** completes the parallel evaluation process.

Overall, Erlang provides a simple but powerful framework within which it is possible to develop concurrent application routines. The **parallel_eval** and **promise** modules provide good abstractions over concurrency, giving the programmer sufficient control of decisions relating to the distribution of arguments.

Appendix 1: Implementation Details Regarding Various Parallel Map Declarations

Appendix 0 provided significant details of distributed Erlang as it relates to list processing. Appendix 1 will examine various parallel map functions, designed and implemented as part this research.

As stated previously in Appendix 0, the crux of **parallel_eval** lies within the collection of arguments given to **parallel_eval** in the form of an argument list. For simplicity, we will call the formation of argument lists, splitting. Erlang provides a simple splitting routine for its parallel library map routine (**pmap**) which returns a list containing the normal Module, Function, Argument List triple but restricts the Argument List to a singleton; that is, of the form:

[{mod_name, function, [term$_1$]}, . . . ,{mod_name, function, [term$_n$]}]

As was discussed earlier in the paper, this approach is rather inefficient, creating an unnecessary amount of overhead. Two other splitting techniques though seemed quite plausible, and admittedly, more efficient.

The first **smap**, splits the arguments into sublists of eight terms, thereby mimicking the serial nature of **pmap**, without the overhead of remote procedure calls with single element lists. The split definition for **smap** is declared in the following way:

```
split_args(M,F,As,List,Len,Ack) ->
   case Len < 9 of
     true -> [{lists,map,[{M,F},As,List]}|Ack];
     false -> split_args(M,F,As,lists:nthtail(8,List),Len – 8,
          [{lists, map, [{M, F}, As lists:sublist(List, 8),]} | Ack])
   end.
```

split_args takes a module **M**, a function **F**, a list **As**, a list **List**, the length of **List** (**Len**), and an accumulator **Ack**. The **case** statement **Len < 9,** stops the recursion, while any list larger then 9 is split on the first 8 terms, appended to the accumulator, and used as the new **Ack** in the (tail) recursive call to **split_args**. The behavior of **smap** dictates that the non-parallel library routine map, will be executed as a spawned process, as opposed to **pmap**, where the actual applied function is spawned.

The second splitting technique used in **tmap**, performs a simple binary split of the input. The declaration is a bit different though. Instead of giving 3-tuples of the form **{lists,map,[foo]}** or **{mod,applied_func,[foo]}** to **parallel_eval** as input, it sends itself, thereby creating a recursive spawning behavior. As a result, the declaration calling **parallel_eval** must include a base case to stop the recursion. We take advantage of this fact, stopping recursive spawns on lists of length eight or less, then choosing the **smap** style of evaluation through the non-parallel library routine map.

tmap, the function which calls **parallel_eval** is defined below:

```
tmap(M, F, As, 0, []) -> [];
tmap(M, F, As, Len, List) when Len < 9 ->
      lists:map({M, F}, As, List);
tmap(M, F, As, Len, List) ->
      lists:append(rpc:parallel_eval(
                   tree_split_args(M, F, As, List, Len))).
```

tree_split_args is defined below:

```
tree_split_args(M, F, As, List, Len) ->
      L_O_L = Len div 2,
      [{newtmap, tmap,
        [M, F, As, L_O_L, lists:sublist(List, L_O_L)]},
       {newtmap, tmap,
        [M, F, As, Len - L_O_L, lists:nthtail(L_O_L, List)]}].
```

Note **div** is Erlang's integer division infix operator, and **lists:sublist(List, n)** gives the first n items of list **List**, while **lists:nthtail(n, List)** provides items **n+1** to end of list.

References

[Armstrong et al. 1996] Armstrong, Joe, et al. (1996). Concurrent Programming in Erlang. Englewood Cliffs, NJ: Prentice-Hall Inc.

[Springer 1989] Springer, George; Friedman, Daniel P. (1989) Scheme and The Art of Programming. Cambridge, Massachusetts: The MIT Press.

[Wilhelm 1995] Wilhelm, Reinhard; Maurer, Dieter. (1995). Compiler Design. Edinburgh Gate, Harlow, England: Addison-Wesley Publishing Company Inc.

[Plasmeijer 1997] Plasmeijer, Rinus; van Eekelen, Marko. (1997). Concurrent Clean Language Report. University of Nijmegen.

Acknowledgments

Great appreciation is given to the University of Minnesota, Morris in regards to financial support for this research through the Morris Academic Partnership (MAP) Program.

Many thanks to Jason Anderson for his helpful work in the University of Minnesota, Morris Computer Science lab (a.k.a. The Dungeon).

Also, a quick thanks to the students of Dian Lopez's Computer Systems and Organization class for putting up with the times when computers were unavailable due to our research.